B4

(54) Title: METHOD AND APPARATUS FOR DEALING WITH DATA CORRUPTION AND SHARED DISKS IN THE CONTEXT OF SAVING, USING AND RECOVERING DATA
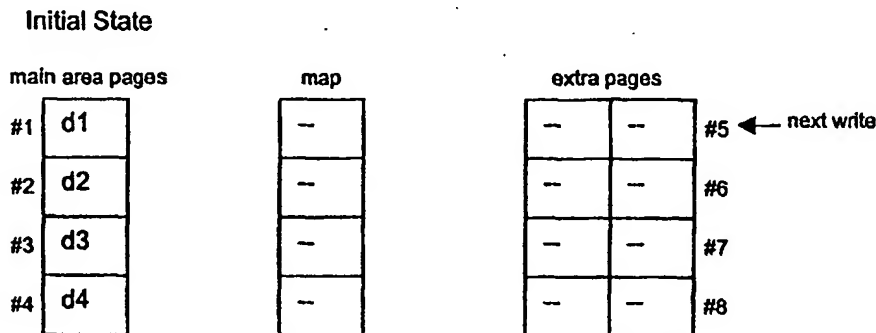
(57) Abstract

    There is described a method of maintaining a record of changes to data on a recording medium and allowing reconstruction of the state of the medium at a prior time. Various alternative embodiments are provided to enhance and improve this process, including maintaining two copies in RAM of data that are compared to detect corruption prior to transferring to the recording medium, splitting logic protection and viewing, disguising the disk to protect against incompatible software, flushing a write cache by insuring a certain amount of time has passed, flushing a write cache by insuring a certain amount of free time has passed and inserting delays if required, inserting safe points after a drop in the rate of disk activity, inserting safe points periodically, flushing the entire OS cache after detecting the need to establish a safe point for one user, separating out user activity by watching directories and files, or the use of complex linked data structures to implement the engine where such data structures can be substantially reconstructed from sequentially organized tables in the event of corruption.

METHOD AND APPARATUS FOR DEALING WITH DATA CORRUPTION AND SHARED DISKS IN THE
CONTEXT OF SAVING, USING AND RECOVERING DATA

5

Copyright Notice/Permission

A portion of the disclosure of this patent document contains material

which is subject to copyright protection. The copyright owner has no objection

to the facsimile reproduction by anyone of the patent document or the patent

10   disclosure, as it appears in the Patent and Trademark Office patent file or

records, but otherwise reserves all copyright rights whatsoever. The following

notice applies to the software and data as described below and in the drawing

hereto: Copyright © 1999, Wild File, Inc. All Rights Reserved.

Incorporation by Reference

15   The entire application of U.S. patent application Serial No. 09/105,733,

filed June 26, 1998, and entitled "METHOD, SOFTWARE AND APPARATUS

FOR SAVING, USING AND RECOVERING DATA" is incorporated herein by

reference.

Technical Field of the Invention

20   The present invention pertains generally to the storage of digital data, and

more particularly to method and apparatus for the backup and recovery of data

stored by a digital computer.

Background of Invention

In one embodiment of the system described in U.S. patent application

25   Serial No. 09/105,733, filed June 26, 1998, and entitled "METHOD,

SOFTWARE AND APPARATUS FOR SAVING, USING AND

RECOVERING DATA" (hereinafter referred to as --U.S. '733--), dealing with

redirecting disk writes to preserve the data being overwritten, involved two

elements: a circular history buffer system and a main map. The history buffer

30   initially received the diverted newly written data while the main map kept track

of actual current location of a given page's data, as viewed by operating system

(unaware of the redirection). In the Temp method, as time permitted, swaps were

performed to exchange the redirected data and historic "overwritten" data into

their appropriate non-mapped hard disk locations. The Always method

minimized swapping by redirecting newly written data to alternate physical disk
locations in which the data is generally left, thus requiring long term re-mapping.

U.S. '733 envisions that in some situations, the swap activity required in
the Temp and Always methods could fall behind the re-cycling of historic data.

5   In other words, the user overwrites so much data that the situation arises where
location X contains a very old piece of historic data HD that is awaiting
swapping to location Y where it really belongs (for optimal organization---i.e.,
near related data), and an overwrite of some location with new data ND occurs.
Since the new data ND is redirected to overwrite the old historic data HD, which

10   would normally be at location Y but is temporarily at location X, U.S. '733
described a process where ND is written to location X and the maps updated
accordingly. This process involves reasonably complex inter-linked maps. U.S.
'733 describes the use of stable and transitional pages to manage inter-linked
maps under the control of a switch page.

15   In a perfect world the techniques of U.S. '733 work very well. However,
real world experience has revealed a potential weakness in U.S. '733. That is,
many personal computers contain buggy code and malfunctioning hardware
which results in data corruption.

Summary of the Invention

20   According to various embodiments of the invention, there is provided a
method of maintaining a record of changes to data on a recording medium and
allowing reconstruction of the state of the medium at a prior time. Various
alternative embodiments are provided to enhance and improve this process,
including maintaining two copies in RAM of data that are compared to detect

25   corruption prior to transferring to the recording medium, splitting logic
protection and viewing, disguising the disk to protect against incompatible
software, flushing a write cache by insuring a certain amount of time has
passed,flushing a write cache by insuring a certain amount of free time has
passed and inserting delays if required,inserting safe points after a drop in the

30   rate of disk activity,inserting safe points periodically,flushing the entire OS
cache after detecting the need to establish a safe point for one user, separating
out user activity by watching directories and files, or the use of complex linked
data structures to implement the engine where such data structures can be

2

substantially reconstructed from sequentially organized tables in the event of
corruption.

<div align="center">Brief Description of Drawing</div>

Figure 1 illustrates the initial state of a New Temp method according to
5   the present invention.

Figure 2 illustrates what happens after the operating system writes the
data "d2a" to location 2 in the example of Figure 1.

Figure 3 illustrates a safe point being established according to one
example embodiment of the invention.

10   Figure 4 illustrates three more writes according to one example
embodiment of the invention.

Figures 5 and 6 show swaps being performed where diverted and historic
data are exchanged and the map updated accordingly, according to one example
embodiment of the invention.

15   <div align="center">The Problem of Data Corruption</div>

The nature of the buggy code problem involves code that is likely not
part of the software implementing U.S. '733 (engine), but other parts of the
operating system, device drivers, or applications. A "bug" could include
intentional maliciousness by viruses. Regardless of the source of the bug, the
20   result is the corruption of the logic (code) and/or data associated with the engine.
Massive corruption does not actually cause the most trouble. Massive corruption
tends to render a computer quickly inoperable, and particularly before the
corruption has propagated from the computer's internal memory (RAM) to disk.
The solution in such a case is to simply restart the computer. Since the disk was
25   not affected, the RAM is re-initialized from the disk's non-corrupted data.
However, a little corruption can wreak havoc that is much more serious. A
common bug in software is a wild pointer (or array index) which results in the
corruption of a few random bytes of RAM (i.e., the location in RAM of the
affected bytes is unpredictable).

30   When code is corrupted, even slightly, the effect is generally
catastrophic---code is a sequence of computer instructions of varying lengths
(number of bytes). If one instruction is "hit" (altered) by the use of a wild
pointer, then not only is this instruction's action altered, but in general, all those

<div align="center">3</div>

that follow it. This is because, if the effect of randomly altering an instruction is
to change its length (the number of bytes making up the instruction), then the
start of the next instruction is improperly located. Thus, if the computer executes
a single corrupted instruction, all the following instructions are typically random

5   junk---parts of other instructions misinterpreted one after another---and the result
is to generally render the current process quickly inoperable. Operating systems
attempt to isolate processes on a computer (using mechanisms built into the
microprocessor's hardware) so that if a process goes awry, its damage can be
contained and the process aborted. A common message under Windows 95 is "a

10   general protection fault has occurred" after which some application or
component of the operating system is terminated. However, if a vital component
of the operating system, such as the engine, is aborted, the entire system for the
most part becomes inoperable. Thus, behavior of a computer after a slight
corruption of the engine's code is much like a massive corruption____when the

15   affected code is called upon (executed), the computer crashes, and likely before
any corruption is propagated to the disk.

Slight Corruption is Worse

The real problem occurs when slight corruption of the engine's data
occurs. This is much more dangerous in that it leads to the engine's logic

20   modifying the disk based on corrupted data (in RAM), and thus propagating the
corruption to the disk. Once this happens, even after restarting the computer, the
engine's data on disk is corrupted. Therefore the essential functions of the
engine, from locating where the user's current data has been mapped to its ability
to represent an image of the disk as it was in the past, are generally lost. This is

25   in part because the techniques of re-mapping the disk jumble the data on disk
and thus without the maps, the data is substantially meaningless.

To make matters worse, the corruption of the engine's data on disk may
not even originate in RAM. In real world practice, corruption can be introduced
in the communication between the computer's main microprocessor and disk

30   controller. The location to write a page or from which to read one can become
garbled, or, the data corrupted in the communication process. Over an extended
period of time, a small amount of data stored on a disk can just go bad (a bad
page). Or, the use of software that is incompatible with the engine, but has the

4

ability to directly access the disk can corrupt the engine's data (such as a disk "doctor" utility).

Again, it is not the massive hardware failures that are a problem. If a computer's hard disk completely fails to function, there is not much that can be

5   done (from a software standpoint). However, at the other extreme, it is unreasonable to allow a single or infrequent glitch to render all the disk's information unusable when in the majority of the disk's data is intact and, in general, the disk is still mostly functioning.

Controlling the Order in Which Data is Written (Write Cache)

10   A final interesting problem inherent in the techniques of U.S. '733 is the need to control the order in which data is written to a disk. For example, when making a transition from one stable state to another, transitional data is written (flushed) out to the disk and then the switch page updated. However, modern disk drives, in an attempt to improve their performance, now include write

15   caches. These write caches buffer up writes and actually commit the data to the disk media in a different order than written. This speeds up the overall write process by allowing, for example, the disk controller to actually write data in an order that reduces the movement of the disk head. However, now the switch page may be updated on disk before data that is assumed already present on disk (it is

20   still waiting to be written). In the event of a power failure, the safe transitioning from one stable state to another is rendered useless.

It turns out that there are commands that can be sent to disk drives to disable such write cache optimization. However, these also disable other useful optimization and thus there is a serious performance degradation. Some disks

25   support the use of a flush command to specifically flush out the write cache, but these commands are not easily available. In other words, on a computer of today, there are standard means in the BIOS to reading and writing from a disk, but there is no standard means to flush the write cache. Thus, regardless of whether a computer's disk drive supports a flush command, since the engine uses the

30   standard interfaces of the BIOS, there is no way for the engine to easily initiate flush. It would have to talk directly to the disk and thus have specific hardware knowledge, which from the point of view of a general program that is expected to run on any computer is not possible. The computer manufacturer generally has

married a specific type of hard disk with a BIOS that knows how to control this type of disk. All software that follows generally relies on the interfaces provided by the BIOS to talk to the disk—be it SCSI, IDE, or other—and the interface today does not include a flush command. Therefore, without attempting to build

5   in specialized disk (hardware) knowledge into the engine, an improved engine design would facilitate the presence of a write cache without requiring a method of flushing it. This implies the engine must take into account that data written to a disk controller may be actually committed to the disk media in a different order, and notwithstanding, the engine must maintain its data integrity on disk at

10   all times allowing for crash recovery.

<u>The Requirements of A Robust Engine</u>

In summary, an engine must elegantly deal with the data corruption, both in RAM and disk, which occurs from various sources in the computer world. It must avoid propagating corrupted RAM data to its disk based data (e.g., wild

15   pointers). It must recover from infrequent corruption that originates with the disk (e.g., corrupted transfers, misdirected writes or reads, or pages that have gone bad). It must maintain stable data structures and be able to recover them in the event of a crash. even if just prior to the crash, only parts of the data that had been written to disk controller actually made it (other parts were lost, never

20   making it out of the write cache). Without special consideration to these issues, the techniques of U.S. '733 can result in complete data loss due to corruption of the essential engine data.

Before continuing on to the present invention, which addresses these issues, one might ask how today's computers manage to generally function in

25   these conditions. The answer lies partially in a program called ScanDisk, which automatically runs whenever a Window's based computer starts and has reason to believe it did not previously shut down in an orderly fashion. Or, the user can manually launch the program if he or she suspects they have "disk problems." ScanDisk examines the disk-based data structures of the operating system and, to

30   the best of its ability, repairs them. The main point is that ScanDisk does not reverse damage or return to some previously known good state, but simply corrects and adjusts various links until the operating system's data structures are again usable. Often, some amount of data is lost or corrupted—i.e., parts of files

are lost or altered or even entire files disappear. However, there are lots of files
on a computer and many are never or seldom used, and therefore a significant
amount of corruption can occur before a computer seriously stops functioning.
Thus, today computers more or less get "sick" in that over time, various parts of
5   the system slowly get corrupted and it is not unheard of for computer users to
periodically wipe their disk clean and re-install the operating system, their
applications, and data.

The Promise of Going Back in Time

This process of gradual corruption is inconsistent with the promises of
10   the engine. The engine's goal is to return the disk to exactly the way it was at
some prior time, and thus must be immune to gradual corruption that does occur,
and which is currently corrected to the best of ScanDisk's ability (or similar
repair utility).

In a more perfect world, an operating system would protect the engine
15   and its data from wild pointers and viruses, and with a little attention to data
redundancy on disk (to protect against an occasional bad transfer), the techniques
of U.S. '733 would suffice. However, operating systems need to be extendable at
their lowest levels, and thus it is hard to make them truly protect one device
driver from another. Further, an operating system is usually a very large program
20   and as such, it is hard to eliminate every bug from within it.

It may seem that all programs would desire "perfect" protection from
another. And in fact the operating system attempts to do so using the hardware
means built into the microprocessor (such as interrupts to take control away from
a program and memory protection mechanisms to trap stray writes). But at the
25   same time, the operating system must balance protection with the need to be
efficient. Efficient code tends not to want to have the burden of protection
boundaries (it takes time to constantly protect one task from another). However,
the engine is not just another program. Its job, when things go awry, is to allow
the computer to restart and everything revert back to some better time. This fall
30   back position must be protected. A user generally perceives a program to run
within the whelm of the computer's RAM and expects, or at least desires, that
upon restarting the computer, the contents of the disk should be intact.
Therefore, it is argued that protecting the engine and disk system warrants

7

special hardware and/or techniques above the compromised level available through the operating system.

The following is a summary of the types of problems an engine must address:

5   • wild pointer data corruption

        • reading corrupted data from the disk (bad transfer, bad page, etc.)

        • use of incompatible software that directly accesses the disk

        U.S. '733 envisioned protecting the engine and its data by special hardware provisions in the main microprocessor (firewall) and/or motherboard,

10   or by actually moving the engine to the computer in the disk controller. This would eliminate the wild pointer problem (assuming the engine was bug free) and prevent incompatible software from directly accessing the disk. However, today there already exist millions of computers for which modifying the hardware is not an option. The present invention offers solutions based on the

15   principles of data redundancy.

Duplication of RAM Data to Detect Corruption

To deal with stray writes through wild pointers (or any other means), the engine of the present invention duplicates all of its major data structures in RAM. For those that correspond to data that is stored on the disk, and cached in

20   RAM, the data's location on disk is included within the data. The critical moment at which corruption can be introduced to the otherwise valid disk image is right before cached data is written back to disk. Therefore, just prior to the write, the two RAM copies are compared to insure they still indicate they are associated with the intended destination on disk, and that their data matches.

25   Should the comparison fail then there is some (RAM) corrupting element at work. Additional techniques could be employed to identify the corruption and correct it, but the best course of action is to alert the user and restart the system. It is felt that once corruption is detected, the system as a whole just cannot be trusted. There is too much chance that further processing will lead to propagating

30   serious corruption to the disk from which there is no recovery. A less drastic response is to block all further attempts to modify the disk, but alert the user and continue to allow the system to run. During this time the user can attempt to save

data to an alternate storage device (e.g., save their word processing document to a floppy).

Considering the fact that computers run using large amounts of complex software provided by many different vendors, it is almost assured that stray

5   writes will occasionally occur. The amount of data employed by an engine, considering that it is re-mapping very large disks, is significant (megabytes of RAM) compared to the microprocessor's address space. Therefore, in the absence of hardware protection, there is a reasonable chance that the engine's data is occasionally corrupted. For example, on a microprocessor with a 32 bit

10   address space (4096M) in which the engine uses two megabytes (2M), there exists a one in two thousand chance that a stray write will hit the engine data. However, the probability that two stray writes will hit the two redundant copies of the engine data, and make the same alternation, and that whatever is causing the corruption will not continue on to do massive corruption, is extremely

15   unlikely.

Once the engine validates the two RAM copies against each other, it writes the data to disk. To insure the data was successfully transferred to the disk controller, the engine could immediately read back and verify the data. Both copies are written out to different locations on disk to add a level of redundancy

20   to the disk based data. Therefore, in the future, should the disk fail to retrieve one copy for any number of reasons, there is a second copy available. Insuring data integrity on disk through redundancy is a well-known principle such as used in RAID systems. However, in this case, the redundancy is limited to two copies located generally on the same physical drive.

25   Splitting up the Engine

The best solution to protecting the engine and its data has been described here and in U.S. '733 is to provide hardware protection and host the engine on the motherboard (using the main microprocessor) or in the disk controller. It has been argued that because of the substantial RAM and processing time

30   requirements, the ideal location for the engine is on the motherboard (and not the disk controller). However, the engine is really performing two somewhat separate functions: it is managing the reuse of historic data and it is re-mapping the disk. It is through re-mapping that the engine presents to the operating

9

system different "views" of the disk, showing the current image as well as states
based on the past. However, should the re-mapping and the views of the disk
become temporarily corrupted, this is acceptable as long as the user can restart
the computer and be (reasonably) guaranteed that the disk and the ability to go

5   back in time is preserved. The reasoning is this: it does not matter whether it is
an application, or the operating system, or even certain parts of the engine that
gets corrupted, if the end result is same---the user sees or experiences unexpected
results, which can only be "cleared" by restarting the computer.

       Thus it is possible to split some of the functions of the engine between an
10   unprotected main processor (and motherboard) and the disk controller. The
memory and processing intensive tasks, like re-mapping, are left to the main
processor (which has the horsepower) while less demanding logic is added to the
disk controller to protect and manage the engine's data structures. The
unprotected part of the engine is called the OSfilter and the portion in the disk
15   controller is the called the enforcer. Thus, the main processor based OSfilter
handling the re-mapping reads data from the disk controller based enforcer, the
information necessary for it to generate and manage the maps used to present
various views to the operating system. As data is written by the operating
system, it is the enforcer that assigns and insures that the new data goes to
20   alternate locations on disk. However, as the new data is redirected, the OSfilter is
informed of the new locations and adjusts its maps accordingly. In the event the
OSfilter's data becomes corrupted, the user can restart the computer and the
OSfilter rebuilds its data from that provided by the enforcer.

       When a disk controller is enhanced to implement the enforcer role, the
25   disk operations that are requested and performed by the disk controller (enforcer)
change from the normal read and write page requests. The read request remains
unchanged, as there is no reason (*) to protect the engine's data and old historic
data from being read. However, the write request now implies that the enforcer
will divert the actual write to another location, and inform the OSfilter of this
30   new location. It is the job of the OSfilter to remember that if the operating
system, in the future, reads the location just written, it must re-map the location
to where the write was diverted. There are various algorithms for determining
where to divert a write. These have been discussed in U.S. '733's Temp and

Always methods. In both cases, it is likely that the initial location to which a

write was diverted will later be updated for performance reasons. This

rearranging of the disk is implemented through a swap process. Since this

process involves reading and writing current and historic data, as well as

5   updating the engine's disk based data structures, it is implemented in the

enforcer.

The user may wish to protect old historic states of the disk from

unauthorized view due to security issues. Confidential information on disk may

have been deleted or encrypted at some time in the past, and it is important to

10   control access to prior to this time.

The OSfilter may require the storing of its maps on disk, though they can

be reconstructed from data provided by the enforcer. Therefore, the enforcer

must provide access to some amount of semi-temporary disk space---i.e., the

OSfilter saves its data structures on disk but in the event that they appear

15   corrupted, they are rebuilt from the enforcer's protected data. A shut down

command informs the enforcer that all of the OSfilter's semi-temporary data has

been written out. In the command the OSfilter supplies the location of its root

page to its data structures for safekeeping. A special read request allows the

OSfilter to obtain "header" information that guides it to the enforcer's read-only

20   data structures from which it can rebuild its maps. This header page includes the

root semi-temporary location supplied by the OSfilter just prior to the last

shutdown, or is returned null if a crash had previously occurred.

In summary, an enforcer based disk controller would accept, among

others, the following basic operations. Additional operations are envisioned such

25   as to support protecting against quickly overrunning the history buffer (i.e.,

insuring some minimum distance into the past is maintained) and locking

historic data relating to one or more SOSD images.

  • read page N, data D returned

  • write data D to location X, diverted location Y returned

30   • request semi-temporary location, location SX returned

  • write semi-temporary data D to location SX

  • release semi-temporary location SX (or all)

  • swap (or move) the contents of locations X and Y

• read header page, data D returned

• disk shut down semi-temporary location SX is root

The enforcer validates the swap requests that are made by the OSfilter, as
the enforcer has knowledge of the applicable data structures. Thus, although the

5   OSfilter can read and reference the data structures generated by the enforcer, the
OSfilter, because it is not trusted, can never make changes or request actions that
would invalidate the enforcer's data structures. In other words, the enforcer
implements the portions of the engine that insure that overwritten OS visible
data is made historic, the historic data is aged and properly recycled, and any re-

10   organization of the disk (for optimization) is done without the possibility of
corrupting its own data structures. The OSfilter, on the other hand, implements
the rest of the engine, such as re-mapping the disk locations supplied by the
operating system to where they are currently located (as controlled by the
enforcer).

15   A disk running in an OSfilter/enforcer mode is unusable by an operating
system without the OSfilter to re-map its accesses appropriately. Therefore,
when a disk controller is in this mode, it is recommended that it not accept the
normal commands (e.g., read or write page). However, it is envisioned that the
disk can switch between normal and OSfilter/enforcer modes. When switching

20   out of OSfilter/enforcer mode, all historic data is discarded and the pages making
up the main image are moved into their non-mapped locations.

Software Protection in the Absence of Hardware Protection

The OSfilter/enforcer split is a hardware-protected implementation of the
engine. As such, it can protect against the use of incompatible software that

25   attempts to directly access the disk. For example, various utilities on the market
change the master boot record (MBR) page. This page must be under the
engine's control in order to insure the engine (or OSfilter) is properly loaded.
Further, if a user boots up on a floppy disk instead of the hard disk, they will not
have loaded the engine (or OSfilter) required to view the hard disk. However,

30   with the enforcer implemented in the disk controller, such incompatible software
cannot damage the engine's data structures.

Although it has previously been stated that updating the hardware for
millions of PCs is impractical compared to releasing new software, it is a less

difficult task to introduce a new hard drive. After all, as capacity increases, hard drives are routinely upgraded on PCs. Therefore, providing a hardware solution in the hard drive coupled with new software that runs on the main CPU (motherboard) is useful. Note the "hardware" solution in the hard drive likely

5   consists of modifying its firmware (software).

In the absence of any hardware protection, the engine disguises the disk to avoid incompatible software modifying the disk. It moves the critical OS data structures, like the FAT and partition tables, to new locations and replaces them with data structures that represent a disguised disk. The disguise takes one of two

10   forms: an unusable disk (corrupted beyond any attempt to repair or modify) or an entirely valid but harmless fake disk. In the latter case, the FAT and partition tables represent a very small yet properly structured "fake" disk that is embedded in the overall disk. Since everything about this fake disk is valid, repair utilities should stay within the boundaries of the fake disk. However, the engine knows

15   where the real data and bounds of the disk lie. Since the engine does not rely on any of the data associated with the fake disk, changes made to the fake disk would not corrupt the engine's data structures.

Duplicating Data on Disk to Avoid Loss

Once engine data has been validated as uncorrupted in RAM and is about

20   to be written to disk, the data is written to two different locations. Thus, if the retrieval of one copy fails, there is a second available. The two copies should be written to somewhat distant locations so that corruption in one area of the disk does not cause the loss of both copies. Further, the actual write process should be separated into two independent write requests to avoid both copies traveling

25   from the motherboard to the disk controller in one request. Once the data has made it to the disk controller, it is likely cached in a pending write buffer and actually written out to the disk media some short time later in an optimal manner (minimizing head seek and transfer time).

In some cases it may make sense to only write one copy of a given

30   engine data structure to disk as long as it is known that this data structure could be rebuilt from other duplicated data structures in the event of a disk error. For example, the main area map can be built from the historic data.

Flushing the Write Cache without a Flush Command

Assuming the engine does not have the ability to control flushing of the write cache, the cache poses a difficult problem in the event of a crash. For example, a power failure likely will cause part of the write cache never to be

5 written to disk. The problem for the engine is that some of its data may have been written to disk, and other parts not; but due to the write cache, the order in which data was written by the engine may not correlate to that which made it to the disk. Thus, if the engine writes to disk locations one, two, and three, and then crashes, it is possible locations one and three have been updated but not two.

10 Since there are few data structures that are fully useful when parts are missing, the present invention uses time aging as a means of insuring the integrity of data written to disk. It is generally assumed that a write cache algorithm will not hold off committing its data to disk for an unlimited amount of time. It is assumed this is true even in situations of constant read and write

15 activity. If this was not the case, then it is conceivable that a file written in the – morning would not make it to the disk media even hours later, because the data is stuck in the write cache awaiting an "optimal" time to be written. The present invention builds upon the assumption that upon writing a page to the disk controller, after some minimum amount of time has passed (*wait time*), it really

20 will be written to the disk media (and so have been saved to non-volatile storage in the event of a crash).

Thus the present invention uses time aging of data, where if and only if the data has been written to the disk controller at least *wait time* seconds in the past, the data is trusted in the context of a recovery (system restart). Some form

25 of timestamp or time marker is included with a block of data that is written to the disk controller, where a block is made up of multiple disk pages. The block is assumed to have been actually been transferred in its entirety to the disk media if a subsequent timestamp or time marker is found that was written at least *wait time* seconds later. A timestamp is a value that directly corresponds to a reading

30 of a clock. By comparing two timestamps, it is possible to determine the length of the represented time interval. On the other hand, a time marker simply indicates that *wait time* seconds have passed since the prior time marker.

Watching the amount of data that has been transferred to the disk controller can be used to approximate a *wait time*. Since the transfer rate to the disk controller is known, multiplying this by the *wait time* yields the amount of data that must be transferred after the write of a given page in order to insure this

5   page has really been written to disk media. This assumes continuous transfers are done.

The assumption that after an appropriate wait time data previously written will actually make it to the disk may not hold for certain disk controllers. In such a case an alternate method is used. Here, the engine insures that there has

10   been a sufficient amount of "free time" since a given write for the disk controller to flush its write cache to disk. The engine can monitor the average number of transfers to the disk controller. This is compared to the known or estimated maximum transfer rate (reflecting the rate at which data can actually be read or written to the disk media) to determine if it is reasonable to expect the write

15   cache to have been flushed. If a flush is required, and an insufficient amount of free time has passed, the engine can simply insert a delay (or a series of short delays to avoid a sudden pause in disk activity). The worst case transfer rate of the drive can be inferred from a timing calibration program that reads and writes very large blocks of data (much more than any reasonable cache size). Note that

20   calculations would have to take into account the number of transfers, their relative proximity, and their size, as each independently contributes to the overall transfer rate. In other words, there is a time overhead to physically moving a disk head associated with the start of each transfer and a certain amount of time spent actually transferring the data to the disk's media.

25   An analogy to the free time method of flushing a disk controller's cache is to imagine the cache as a bucket in which cups of water are added. The bucket has a hole in its bottom that allows it drain at a fixed rate. The water flowing out of the hole is equivalent to the disk controller writing data to the disk media. The process of adding cups of water is equivalent to writing data to the disk

30   controller. If the bucket is full, then you have wait. Reading data from the disk controller is equivalent to momentarily plugging the hole---nothing drains, nothing is added, but time simply passes (it is assumed that a read cache is independent of a write cache). Now, the situation of flushing a write cache is

15

much like needing from time to time insure that a given cup of water that has been added to the bucket has drained, but you cannot actually look inside the bucket or detect if water is still draining. Further, when you add a cup of water to the bucket it gets mixed in with those cups previously as well as those that are

5   subsequently added.

The only way to really know that the cup of water in question has really drained is to insure that at some point after adding the cup, the bucket has completely drained for a moment. Determining when, or how to create this event is the challenge, since you cannot see inside the bucket or know if it is still

10   draining. In other words, there is no standard interface to ask a disk controller if its cache has been flushed. The solution is to determine the worst case rate at which water drains from the bucket. This is done by a test in which you add cups of water as fast as possible until you are forced to wait, since the bucket has become full. After this point, you continue to add cups as the bucket empties

15   (and will accept another cup) and you measure the rate. This process has effectively defeated the "buffering" effect of the bucket and you are now . measuring the actual drain rate (or rate at which the disk controller can empty its cache).

With the knowledge of the rate at which the bucket drains, you can

20   monitor the rate at which you add cups of water. When you add a cup that you wish to insure has been drained, you need to slow the rate of future additions such that the bucket is draining faster than you are adding new cups. At some . point that can be calculated, the bucket will completely drain, notwithstanding the fact that you are continuing to add cups, be it slowly. Of course, if you need

25   to relatively immediately insure the bucket has drained before adding even one more cup, you can simply wait the calculated time it takes to drain the bucket. In fact, by continuously monitoring the rate of additions to the bucket, one can even have some sense of how full the bucket was at the time of adding the cup that need to be drained. This allows you to reduce the delay. Of course, in this

30   environment of approximation, plenty of fudge must be added (e.g., if you think it takes one second to completely drain, then you wait two seconds to cover for any slight error in calculations).

This water bucket analogy represents the free time method of flushing a write cache. Note that the rate at which the bucket empties is equivalent to how fast the write cache is flushed, which is a function of the seek time and transfer rates. Therefore the disk proximity of the data in the write cache---the locations

5   of where it should all be written on disk---greatly impacts the time it takes to flush the cache. In other words, to write a lot of data to one region on the disk media takes far less time then writing small bits of data across the entire media. However, it is the latter case that can be timed and used as a worst case flush rate, acknowledging that the actual rate in general will be much better. Since one

10 cannot guess what data is written first from the cache by a disk controller, it is not possible to monitor the page locations written to the disk controller and estimate the seek time to transfer time ratio, and thereby dynamically generate an estimated current flush rate.

Returning to the issue of the disk controller, the next question is when,

15 generally, would a delay be inserted to insure the write cache is flushed. There is the process of diverting writes. In general, the user can only select points in time to access at which there was a lull in disk activity (safe point). This avoids accessing data from the past that was in the process of being modified. Thus as long as the lull is sufficiently long, the flushing of the write cache is inherent in

20 the identification of safe points. When restarting a computer after a crash, the user has the option of reverting to a prior time, which will be a safe point, or simply using the disk in the state just prior to the crash. In the later case, the engine is recovering its data structures to a point in time where there may not have been a lull in disk activity. In other words, the crash occurred in the middle

25 of a long sequence of disk activity. If the *wait time* based flushing method is used, all data written up to *wait time* seconds before the crash will be present. However, if the disk controller cannot be trusted to actually write data to the disk media after *wait time* seconds, then the engine has no recourse but to discard all data written prior to the last recorded flush time marker in disk activity. In other

30 words, as the engine determines that a sufficient amount of free time has passed to insure a block of data will have actually been transferred to the disk media, the engine writes out an appropriate time marker. Once this marker has made it to the disk media, the block's validity is established. On recovering from a crash,

17

data that was written to the media but for which no subsequent time marker was found is discarded as parts of the data may be missing. In practice, a user is generally not interested in data that was in the process of or had just been written out at the time of a crash, and so discarding of this data is not a problem.

5   Another type of engine activity that requires flushing is when the contents of the disk are being rearranged (swapped). This process involves distinct transfer steps that are assumed, at specific points in time, to be flushed to the disk in order to make the process crash proof (the process can resume in the event of a crash without loss of data). It is possible there is a lot of data to move

10   and thus many points at which flushing must be done. Thus, it is during this activity that delays might be required at specific points to insure that writes have actually made it to the disk.

A simple example of a reasonably efficient swap process would involve the use of a fifty-megabyte swap region. At one megabyte per second average

15   transfer rate, it would take fifty seconds to write all this data. If the disk controller had a one-megabyte write cache then it would take one second of free time for the cache to be flushed to disk. Therefore, at some point after writing the fifty-megabyte swap region, a (cumulative) one-second delay in disk transfers to the disk controller is inserted in order to insure the one megabyte cache is

20   flushed (at one megabyte per second). The ratio of fifty seconds of swap region writing to one second of delay is reasonable (a two percent performance hit).

Note that an important aspect to making this ratio reasonable is that the amount of data written between points at which it must be known flushed to the disk can be large (in the last example, fifty-megabytes was used). On the other

25   hand, if only one megabyte was to be written before a flush was required, then the ratio between time spent transferring data and inserted delays becomes much less desirable (one second of writes followed by a one second delay, or a 50% performance hit). In any actual implementation, the engine must generously fudge up the delay times, as exact timing of a disk controller is hard to know.

30   The need to have large areas that are written between (effective) flushes dictates an engine design where a given disk location must not require modification, followed by flushing, followed by modification at a rate faster than writing the large areas between flushes, as just described. More about this later, although

18

note that the swap process described in U.S. '733 entailed frequent flushing, the need of which has now been overcome by the present invention.

Swapping is generally done in the background and is interruptible by user requested disk activity. If in the process of swapping, a free time based flush is
5   required, and the user starts a burst of disk activity, the required delay is simply inserted after the user's activity completes.

Other points in time at which the write cache must be flushed are when the history buffer is being rapidly overwritten and the diversion process is suspended and when the system is shutting down. Neither of these events occur
10   with any frequency and thus adding some amount of delay is of no consequence.

The actual method of indicating on disk when a block of data is valid is to write out a concluding flag, such as a timestamp or time marker, after a sufficient *wait time* or amount of disk activity free time has passed.

Constant Disk Activity and Safe Points

15   The concept of a safe point is that of a point in time at which everything that reasonably belongs on disk is present (and actually written to the media). This means that any disk based operation, such as saving a file, has completed and all the supporting operating system structures have been updated. At such a point, the system could be abruptly stopped (reset or power removed) and the
20   user would expect to find their data intact upon restarting the computer. In general, the engine uses a lull in disk activity to mark such a point. (Of course, a signal from the operating system to the engine would also suffice.) In a situation where a computer is reset in the middle of heavy disk activity, one cannot count on what has actually been written to the disk media and thus a user typically
25   understands that "all bets are off." There is, however, another situation. This occurs when there is a relatively constant but light amount of disk activity. Such occurs, for example, when downloading a large file using a low speed modem. Further, this activity may be occurring generally at the same time as other activity like the saving of a word processing document (i.e., a user can download
30   and edit at the same time).

In the context of light but long duration disk activity the use of a simple lull (in disk activity) to establish safe points will fail to establish any. This is a serious problem not so much for the data that was being written slowly, but for

19

any unrelated and independent updates that are intermixed. In other words, consider the case of a four hour download, and all the while a user is editing a repeatedly saving a certain file. If their computer crashes, it is important to provide safe points such that much of the download as well as the different
5   versions of the certain file are recoverable.

The present invention therefore employs a more sophisticated means of detecting when safe points should be inserted. Instead of looking for a complete lull in disk activity for some minimum duration, the engine monitors the rate of disk activity. Whenever there is a sudden drop in activity a safe point is inserted.
10   Also, as long as the disk activity rate is below that reflecting a maximum level (i.e., the rate is such that the operating system is likely frequently flushing out its cache), safe points can be automatically inserted (forced) at a reasonable rate. Viewing the disk as it was at a forced safe point will likely yield much of the data written up to the selected time, though some file updates may have been
15   caught midstream.

Forced safe points can also be created after observing a sudden drop in the rate of file open and closes operations processed by the operating system. In either case, forced safe points should be periodically inserted, after any disk write activity, at some minimum time interval to insure some fall back positions
20   (safe points) are always available.

Shared Disks

The prior discussion of safe points assumed a single user computer system where the engine can infer the user's activity from disk activity. For example, when there is a burst of disk write activity, the user is likely saving a
25   file. If you go to the time prior to the burst and view the disk, you see the file before it was updated, and after the burst, you see the file as it was changed. However, in a shared disk environment (e.g., a network file server), the activity of the two or more users is blended together in the overall disk activity. There is no way to identify system wide safe points, as a lull in one user's disk activity
30   may occur in the middle of another user's activity. Thus, in order to support multiple users, there must be some means of separating out each user's activity and insuring it is flushed to the disk after their activity ceases. Further, it is not desirable to allow one user's heavy use of the disk to greatly reduce the recovery

ability for all users. Thus, from the user's standpoint, it is desirable that each user has their own independent history buffer.

A major difference between a single user computer system and a shared disk system is that reverting a single user system back in time is often a good
5   solution to solving a problem that has recently been introduced, but whose nature is not really understood. In other words, if a user's computer is not working now, but was working a hour ago, the easiest course of action is to revert to an hour ago and then recover any desired files that were modified during this discarded hour. However, in the context of a shared disk, one would not want a user to
10   revert the entire system back in time to solve a problem only they are experiencing. In a shared disk environment generally the system is working properly and users have the need to recover specific files or directories. However, if a system, single or shared by many users, gets corrupted such that it no longer functions, reverting the entire system back in time is the good solution.
15   In a shared environment, it assumed the operating system periodically flushes its internal data structures describing the organization of the disk (from RAM to the disk) so that reverting to some point back in time yields a substantially usable system. Under Windows NT, for example, the engine can initiate the flushing the OS cache (the operating system cache being different than the disk controller's
20   cache) to insure that this process occurs at some desirable rate.

Reverting a shared disk to some point in the past, which affects all users, is useful primarily when the system has a whole has been rendered inoperable. This may occur, for example, when major software upgrades or updates are performed.

25   Outside of a shared disk system becoming inoperable, the act of reverting the entire disk is not useful. Instead, users may wish to revert (restore) certain files and directories. This leads to the need to separate the user's activity and insure their data is flushed from OS cache. In other words, take the example that one user modifies a file and the changes go into the OS cache, and then another
30   user does a lot of file I/o such that the OS cache is not flushed, and then the original user again modifies the file. All of this activity will never have been written to the disk and therefore there is no way to restore, for example, the first version of the modified file based on what was written to the disk.

21

The solution is for the engine to monitor the users' activity, separating one user from another, and when it appears that a given user has finished a series of file changes, these changes are forced to be flushed from the OS cache. It may turn out that it is difficult to separate out what data in the OS cache is associated

5   with which user, particularly considering some data may have been modified by more than one user. A solution is to simply flush the entire cache at points corresponding to the need to establish a safe point for one user, even if other users are in the middle of making changes.

The next question is how does one isolate file activity of one user from

10   another. If the engine hooks into the operating system at a level such that it sees each file request and the user from which it originates, it can easily look for lulls in activity independently for each user. When a lull is detected, the entire OS cache can be flushed and a safe point for the specific user established.

If there is no way to hook into the operating system at a level to associate

15   individual users with the file requests, then the next best method is to watch the frequency of access to files and directories. One could look for a lull in access to non-shared directories. One could also keep a list of files that have been modified. If some minimum time has elapsed since the file was last modified, and now another request to modify the file occurs, the OS cache could be flushed

20   and a safe point established before the new request is processed. This would insure the file's prior state could be restored, if required. This forced safe point would naturally only be generated when multiple users' activity is overlapping. If a user's activity occurs as a burst without overlapping another user's activity then the flushing and creation of a safe point occurs naturally in the lull that

25   follows the user's burst of activity (i.e., this is the single user scenario). If so many files are being modified that the engine cannot keep track of them all, it can simply force flushing and a safe point, and start over with its tracking. In other words, it never hurts other than in performance, to have more safe points than required. On the other hand, it is important not to miss a safe point, since

30   this implies the user cannot return to the corresponding point in time.

The separation of users' activity using the method of forcing safe points when a specific file is about to be modified a second time, and after some minimum time has elapsed since the first modification, is indirect. The engine is

simply insuring that safe points exist to reasonably separate out the different versions of files, which implies the separation of users' activity. There will be more safe points than required for any individual user, but each user will pick out from the overall set of available safe points those associated with their own

5   activity.

A Simple Reconstructible Mapping System

Returning to the issue of corruption of the engine's data structures, U.S. '733 described (though did not mandate) the use of complex data structures for which the loss of some of the data can lead to the loss of all the data. For

10   example, if a tree data structure is used to re-map the disk and a portion of the tree (node) is lost, the entire tree can be rendered unusable. Since the tree describes where the user's data is really located on disk, without the tree the user's data is effectively lost (or more precisely, scrambled).

U.S. '733 described the use of history headers and a main area map.

15   Although the Temp method in U.S. '733, as time permitted, swapped historic and main area data into their appropriate locations, it was possible that unswapped historic pages would be reused. They could contain new main area data, or internal engine maps, or basically most any type of data managed by the engine. Consider the case of a disk with some location, say 10, containing the

20   data AA corresponding to the OS visible location 10 (no mapping required). Now the user overwrites this data with BB and the engine diverts the data from location 10 to 50. Location 10 now holds the historic data AA. In time, the historic data AA becomes very old as more data is modified and new historic states maintained. Eventually location 10 becomes the oldest historic data and

25   therefore the next location to re-use for whatever purpose is needed by the engine, such as a node in a map, or to receive a diverted write. The point of this scenario is to illustrate how the described Temp method leads to the storing of all sorts of data types across a disk's media (i.e., intermixed with the user's current image data).

30   Now we introduce the problem of uncorrectable corruption, caused by some unknown source. The techniques previously described how duplicating data on disk and in RAM, and careful validation to detect RAM corruption before it propagates to the disk will substantially reduce the likelihood of

corruption, but it may occur anyway. If the data structures involve links, where
for example, page A contains data that points B, and B contains data that points
to C, the loss of page B's data means that C is also lost. Thus, the corruption of a
complex link-based data structure is generally catastrophic.

5   On the other hand, a data structure that is naturally resistant to partial
corruption is a sequential sequence of pages. For example, if page locations #5,
#6, #7, and #8 contain the ASCII text of a document, and page #6 gets corrupted,
one can still retrieve part of the text from page locations #5, #7, and #8. On the
other hand, if part of the data in page #5 indicated the location of the next page,
10   and this page was found to be corrupted, there would be generally no way to
retrieve the rest of the document.

        Therefore, the present invention adjusts the data structures and processes
described in the Temp method based. Figure 1 illustrates the initial state of a
New Temp method. Here you see a set of main area pages, a map, and a set of
15   extra pages and history headers (the two columns located right next to each
other). Figure 2 illustrates what happens after the operating system writes the
data "d2a" to location 2. The write has been diverted to an extra page and a note
made in the corresponding history header indicating where the data belongs
(location 2). Also note the main map has been updated to indicate where the
20   operating system's location #2 is really located. In Figure 3 a safe point is
established. Figure 4 illustrates three more writes. Note that one can directly
overwrite data and not preserve prior states when a page is modified that is part
of the current group being formed, as bounded by safe points. In Figure 4
location #1 is written twice with the final state being saved in the extra page
25   areas. Figures 5 and 6 show swaps being performed where diverted and historic
data are exchanged and the map updated accordingly. (The pages involved in any
given swap cycle do not have to correspond to safe point boundaries.)

        At first glance these data structures and processes are much the same
those described in the Temp method. However, this New Temp method does not
30   allow the next write location in the extra page/history header tables to advance
into unswapped data. Thus, if the swapping in Figures 5 and 6 did not occur, the
engine could not continue the algorithm just described. It is hoped that there are
sufficient extra pages and corresponding history headers to absorb any burst of

write activity, with subsequent time available to perform swapping, so that the overrun condition is rare. However, if an overrun does occur where so much data is rapidly written (and diverted) that there are no available post swapped extra pages then the engine shuts down. This involves informing the user (if possible)

5   that all historic data is about to be lost and then proceeding to write data to wherever the main map last indicated. Eventually, when free time does occur, swaps are performed and then the history headers reset (re-initialized to indicate there are no saved historic states).

The New Temp method differs from the original (old) in that it can be

10   forced to shut down. The original method could go on and on, without any swapping, with a form of fragmentation occurring. In time, as the user continues to write without free time for swapping, the writes are diverted to increasingly fragmented locations. The reality of this performance hit was such that the original method would also do a shut down, preferring the limit fragmentation

15   and responsiveness at the cost of giving up tracking historic states. The argument was that if the user is rapidly changing so much data that only very recent changes can be reversed, this is of little use. Therefore it is better to shut down and run faster.

Given that both the new and old Temp methods shut down, the difference

20   is that the new method shuts down sooner. However, the non-wrapping of the next write location into unswapped data simplifies the data structures. Now, if at some point the main map is lost, it can be easily reconstructed from the history headers. In the old Temp method, the main map could come to indicate that two main area pages are in each other's desired location, and thus require swapping.

25   Because of this condition, one could not lose the main map without losing vital swapping information. However, this condition cannot occur in the new Temp method and thus the main map is simply an optimized method of determining from the history headers where a given OS visible page is really located, and so can be rebuilt from the headers.

30   The recovery process in the event the main map is lost simply involves locating the last position of the next write pointer, resetting the main map, and cycling through the history headers. As each header is processed, the OS visible location that is being mapped to its corresponding extra page is add to the main

map. Depending on the direction of movement through the headers, duplicates are either taken, overwriting the prior link in the main map, or discarded. For an example, return to Figure 4 and discard all the links in the main map. Start at the next write pointer and move upward. The first encountered history header

5   indicates location 2 should map to "d2b" and the next indicates location #1 should map to "d1b". The last header is a repeat of location #2, and since because we are moving backward in time through the headers, the duplicate is ignored since the main map reflects the most recent user writes. The main map has now been rebuilt. In fact, this example illustrates that the history headers

10   could play the role of a main map, in that for each OS visible disk access the map could be scanned to see where the reference should really be directed. However, scanning a huge table for every disk access is highly inefficient and therefore calls for constructing a second more optimized lookup means.

The main map is likely a complex linked data structure (like a tree) and is

15   discarded and rebuilt upon finding corruption. If a portion of the history headers is corrupted, then the result is either lost historic states or the loss of recently written new data (which had not yet been swapped with its overwritten historic data). In either case, assuming the amount of corruption is limited, it is very likely that most of the user's data is intact and usable, and much of the historic

20   states are still usable. In the case where unswapped pages are lost the user can be informed that corruption has occurred, and he/she should validate and backup or restore any desired files. The condition of corruption indicates the user's computer has serious problems.

The sequential or table like nature of the extra pages and history headers

25   facilitates substantial reconstruction, and virtually eliminates the likelihood of a catastrophic user data loss due to the loss of key portions of a complex map.

The sequential nature of how the history headers and extra pages are updated as the user writes data easily fits into the previously described method of time aging data to insure it has been flushed through any write cache. In other

30   words, as the user makes changes, data is written following and advancing the next write pointer into the history headers and corresponding extra pages. It is easy to embed time aging notes to identify any areas that were in the process of being flushed from the write cache at the time of a crash.

26

The other major process of the engine, the reorganization or swapping of data on the disk can also be easily implemented to support time-aged flushing. Instead of providing a single swap area and using a switch page to control transition states, a table of swap areas is established. In this way, a series of swap

5   cycles can be initiated, and while data for one phase of a pending swap cycle is being time-aged (flushed and stabilized on the disk), the setup for another swap cycle's current phase can be in progress. If delays must be inserted to accomplish flushes, the use of multiple swap areas at least allows the ratio of flushes to disk activity to be kept optimal (less flushes). The swap phases involve collecting all

10   the data involved in a swap in a swap area and then once this backup copy is safe on disk (and flushed), the data can be moved into place without risk of loss in the event of a crash. See the section on flushing the write cache for more details.

The principles just outlined for the Temp method to address recoverability can just as well be applied to the Always method.

15

Claims:

1.   A method comprising:

maintaining a record of changes to data on a recording medium; and

maintaining two copies in RAM of data that are compared to detect

corruption prior to transferring to the recording medium.

2.   A method comprising:

maintaining a record of changes to data on a recording medium; and

splitting logic protection and viewing.

3.   A method comprising:

maintaining a record of changes to data on a recording medium; and

disguising the disk to protect against incompatible software.

4.   A method comprising:

maintaining a record of changes to data on a recording medium; and

flushing a write cache by insuring a certain amount of time has passed.

5.   A method comprising:

maintaining a record of changes to data on a recording medium; and

flushing a write cache by insuring a certain amount of free time has

passed and inserting delays if required.

6.   A method comprising:

maintaining a record of changes to data on a recording medium; and

inserting safe points after a drop in the rate of disk activity.

7.   A method comprising:

maintaining a record of changes to data on a recording medium; and

inserting safe points periodically.

28

8.   A method comprising:

maintaining a record of changes to data on a recording medium; and

flushing the entire OS cache after detecting the need to establish a safe point for one user.

9.   A method comprising:

maintaining a record of changes to data on a recording medium; and

separating out user activity by watching directories and files.

10.   A method comprising:

maintaining a record of changes to data on a recording medium;

and the use of complex linked data structures to implement the engine where such data structures can be substantially reconstructed from sequentially organized tables in the event of corruption.

11.   Apparatus comprising:

means for maintaining a record of changes to data on a recording medium; and

means for maintaining two copies in RAM of data that are compared to detect corruption prior to transferring to the recording medium.

12.   Apparatus comprising:

means for maintaining a record of changes to data on a recording medium; and

means for splitting logic protection and viewing

13.   Apparatus comprising:

means for maintaining a record of changes to data on a recording medium; and

means for disguising the disk to protect against incompatible software.

14.   Apparatus comprising:

        means for maintaining a record of changes to data on a recording medium; and

        means for flushing a write cache by insuring a certain amount of time has passed.

15.   Apparatus comprising:

        means for maintaining a record of changes to data on a recording medium; and

        means for flushing a write cache by insuring a certain amount of free time has passed and inserting delays if required.

16.   Apparatus comprising:

        means for maintaining a record of changes to data on a recording medium; and

        means for inserting safe points after a drop in the rate of disk activity.

17.   Apparatus comprising:

        means for maintaining a record of changes to data on a recording medium; and

        means for inserting safe points periodically.

18.   Apparatus comprising:

        means for maintaining a record of changes to data on a recording medium; and

        means for flushing the entire OS cache after detecting the need to establish a safe point for one user.

19.   Apparatus comprising:

        means for separating out user activity by watching directories and files.

20. Apparatus comprising:

   means for maintaining a record of changes to data on a recording

medium; and ·

means for the use of complex linked data structures to implement the engine

where such data structures can be substantially reconstructed from

sequentially organized tables in the event of corruption.

21. A compuer readable medium storing a computer program which when

   executed in a suitably configured computer performs the method of claims 1,

   2, 3, 4, 5, 6, 7, 8, 9 or 10.
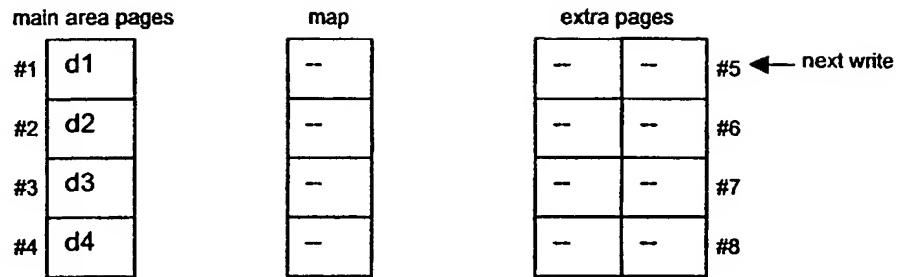
Figure 1: Initial State

main area pages	map	extra pages		
#1 d1	—	—	—	#5 ← next write
#2 d2	—	—	—	#6
#3 d3	—	—	—	#7
#4 d4	—	—	—	#8

Figure 2: Write #2 (d2a)

main area pages	map	extra pages		
#1 d1	—	d2a	#2	#5
#2 d2		—	—	#6 ← next write
#3 d3	—	—	—	#7
#4 d4	—	—	—	#8
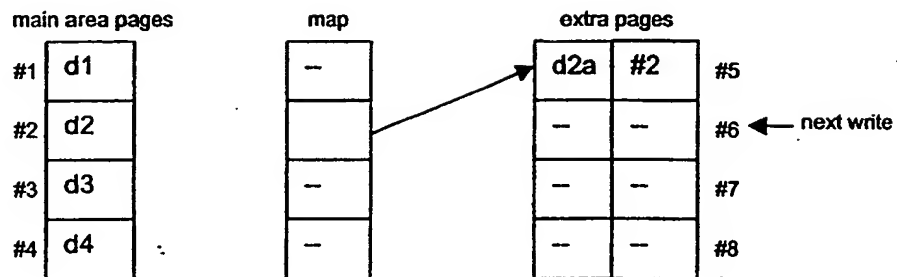
Figure 3: End of write session (safe point)

main area pages	map	extra pages		
#1 d1	—	d2a	#2	#5
#2 d2		—	—	#6 ← next write
#3 d3	—	—	—	#7
#4 d4	—	—	—	#8
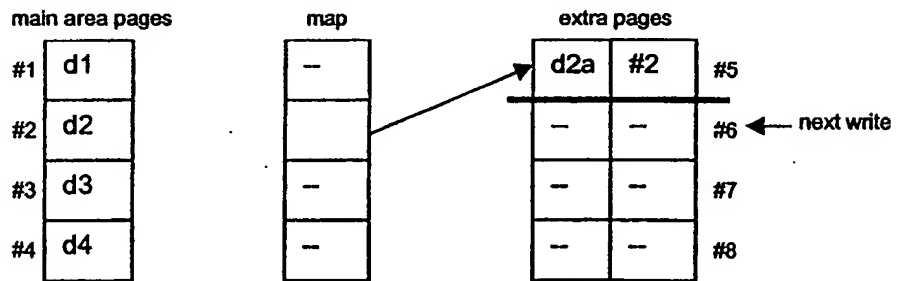
Figure 4: Write #1 (d1a), #2 (d2b), #1 (d1b), and safe point

main area pages		map		extra pages		
#1	d1			d2a	#2	#5
#2	d2			d1b	#1	#6
#3	d3	—		d2b	#2	#7
#4	d4	—		—	—	#8 ← next write
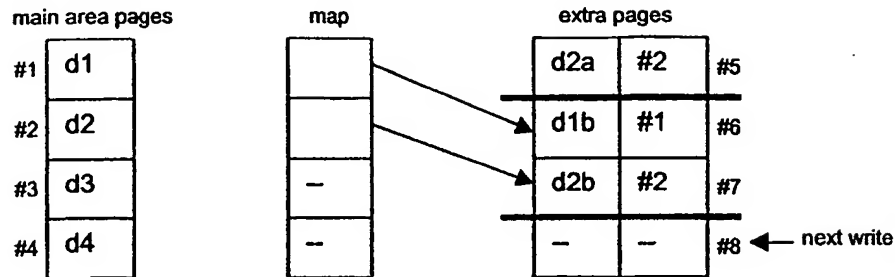
Figure 5: Swap first block (#5)

main area pages		map		extra pages		
#1	d1			d2	—	#5
#2	d2a			d1b	#1	#6
#3	d3	—		d2b	#2	#7
#4	d4	—		—	—	#8 ← next write
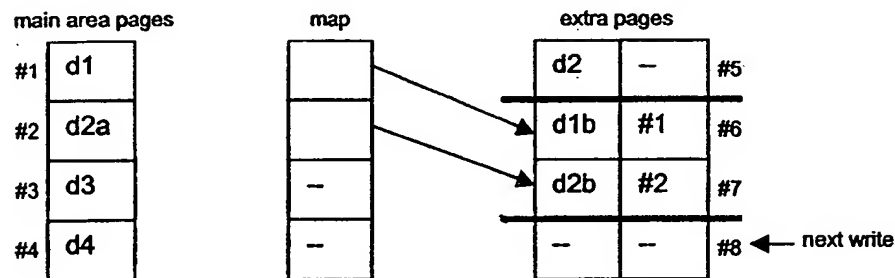
Figure 6: Swap second block (#6 and #7)

main area pages		map	extra pages		
#1	d1b	—	d2	—	#5
#2	d2b	—	d1	—	#6
#3	d2	—	d2a	—	#7
#4	d3	—	—	—	#8 ← next write
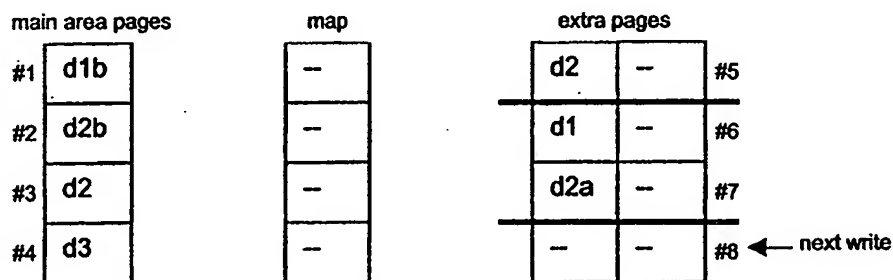
Int   tional Application No

PCT/US 00/10999

A. CLASSIFICATION OF SUBJECT MATTER

IPC 7   G06F11/14

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

IPC 7   G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5 255 270 A (YANAI ET AL.) 19 October 1993 (1993-10-19) abstract	1
A	US 5 893 140 A (VAHALIA ET AL.) 6 April 1999 (1999-04-06) column 17, line 7 —column 18, line 16	4,5
X	WO 99 12101 A (WILD FILE, INC.) 11 March 1999 (1999-03-11) page 115, line 28 —page 116, line 13	6-8
A	US 5 717 849 A (BRADY) 10 February 1998 (1998-02-10) abstract	9

-/--

| X | Further documents are listed in the continuation of box C. | | X | Patent family members are listed in annex. |

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
25 August 2000	01/09/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31–70) 340–2040, Tx. 31 651 epo nl, Fax: (+31–70) 340–3016	Corremans, G

Form PCT/ISA/210 (second sheet) (July 1992)

1

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
X	WO 98 26353 A (HELIX SOFTWARE CO.) 18 June 1998 (1998-06-18) page 7, line 8 - line 16	10

INTERNATIONAL SEARCH REPORT

Information on patent family members

Patent document cited in search report		Publication date	Patent family member(s)		Publication date
US 5255270	A	19-10-1993	NONE		
US 5893140	A	06-04-1999	US 5948062 A US 5944789 A		07-09-1999 31-08-1999
WO 9912101	A	11-03-1999	US 6016553 A AU 9383298 A		18-01-2000 22-03-1999
US 5717849	A	10-02-1998	EP 0682313 A JP 2846837 B JP 7311660 A		15-11-1995 13-01-1999 28-11-1995
WO 9826353	A	18-06-1998	AU 5899498 A		03-07-1998